Linear Algebra Assignment 1 Report

110062219

October 18, 2022

Contents

In this assignment, we are to discover the basic of `numpy` & `scipy`, and BLAS & LAPACK.

1 Configurations

Below are my configurations of `numpy` and `scipy` respectively. I adopted the Intel® Distributed Python environment on the cluster supercomputer **Taiwania 3** of NCHC *(National Center for High-performance Computing)*, which contains Intel® MKL *(Math Kernel Library)*.

1.1 numpy

```
blas_mkl_info:
    libraries = ['mkl_rt', 'pthread']
    library_dirs = ['/opt/ohpc/Taiwania3/pkg/intel
       /2022.2/intelpython/latest/lib']
    define_macros = [('SCIPY_MKL_H', None), ('
       HAVE_CBLAS', None)]
    include_dirs = ['/opt/ohpc/Taiwania3/pkg/intel
       /2022.2/intelpython/latest/include']
blas_opt_info:
    libraries = ['mkl_rt', 'pthread']
    library_dirs = ['/opt/ohpc/Taiwania3/pkg/intel
       /2022.2/intelpython/latest/lib']
    define_macros = [('SCIPY_MKL_H', None), ('
       HAVE_CBLAS', None)]
    include_dirs = ['/opt/ohpc/Taiwania3/pkg/intel
       /2022.2/intelpython/latest/include']
lapack_mkl_info:
    libraries = ['mkl_rt', 'pthread']
    library_dirs = ['/opt/ohpc/Taiwania3/pkg/intel
       /2022.2/intelpython/latest/lib']
    define_macros = [('SCIPY_MKL_H', None), ('
       HAVE_CBLAS', None)]
    include_dirs = ['/opt/ohpc/Taiwania3/pkg/intel
       /2022.2/intelpython/latest/include']
lapack_opt_info:
    libraries = ['mkl_rt', 'pthread']
    library_dirs = ['/opt/ohpc/Taiwania3/pkg/intel
       /2022.2/intelpython/latest/lib']
    define_macros = [('SCIPY_MKL_H', None), ('
       HAVE_CBLAS', None)]
    include_dirs = ['/opt/ohpc/Taiwania3/pkg/intel
       /2022.2/intelpython/latest/include']
```

```
Supported SIMD extensions in this NumPy install:
    baseline = SSE,SSE2,SSE3,SSSE3,SSE41,POPCNT,SSE42
    found =
    not found = AVX512_ICL
```

1.2 scipy

```
lapack_mkl_info:
    libraries = ['mkl_rt', 'pthread']
    library_dirs = ['/opt/ohpc/Taiwania3/pkg/intel
       /2022.2/intelpython/latest/lib']
    define_macros = [('SCIPY_MKL_H', None), ('
       HAVE_CBLAS', None)]
    include_dirs = ['/opt/ohpc/Taiwania3/pkg/intel
       /2022.2/intelpython/latest/include']
lapack_opt_info:
    libraries = ['mkl_rt', 'pthread']
    library_dirs = ['/opt/ohpc/Taiwania3/pkg/intel
       /2022.2/intelpython/latest/lib']
    define_macros = [('SCIPY_MKL_H', None), ('
       HAVE_CBLAS', None)]
    include_dirs = ['/opt/ohpc/Taiwania3/pkg/intel
       /2022.2/intelpython/latest/include']
blas_mkl_info:
    libraries = ['mkl_rt', 'pthread']
    library_dirs = ['/opt/ohpc/Taiwania3/pkg/intel
       /2022.2/intelpython/latest/lib']
    define_macros = [('SCIPY_MKL_H', None), ('
       HAVE_CBLAS', None)]
    include_dirs = ['/opt/ohpc/Taiwania3/pkg/intel
       /2022.2/intelpython/latest/include']
blas_opt_info:
    libraries = ['mkl_rt', 'pthread']
    library_dirs = ['/opt/ohpc/Taiwania3/pkg/intel
       /2022.2/intelpython/latest/lib']
    define_macros = [('SCIPY_MKL_H', None), ('
       HAVE_CBLAS', None)]
    include_dirs = ['/opt/ohpc/Taiwania3/pkg/intel
       /2022.2/intelpython/latest/include']
```

2 Naming convention for BLAS and LAPACK

2.1 What are the naming convention, starting character S, D, C, Z, for BLAS and LAPACK?

Since Fortran used to have strict limit on the length of name of subroutines, BLAS and LAPACK had to name their subroutines in short.

S Single-precision floating-point number

D Double-precision floating-point number

C Complex number

Z Double-precision complex number

The first letter stands for the variable type shown above, the next two ones indicates the type of matrix, and the remains are the abbreviation of the operation.

2.2 What are the meaning of SSPR2, ZGERC, DGBSVX and CHEEVR?

The first two ones are from BLAS while the last two ones are from LAPACK.

SSPR2 Symmetric packed rank 2 operation $A \mathrel{+}= \alpha(xy^T + yx^T)$ for single-precision floating-point number

ZGERC Rank 1 operation $A \mathrel{+}= \alpha x\overline{y^T}$ for double-precision complex number

DGBSVX Computes the solution to system of linear equations $AX = B$ for *General-Band* matrices.

CHEEVR Computes the eigenvalues and, optionally, the left and / or right eigenvectors for *Hermitian* matrices.

3 Find the subroutine to solve a linear system $Ax = b$ in LAPACK, where A, b are in double precision

The **DGESV** *(solve for double-precision general matrix)* subroutine from LAPACK has capability of solving the linear system like $Ax = b$. It applies LU

decomposition to the matrix to perform Gaussian elimiation so as to find the solution.

In the `scipy`-wrapped version **`DGESV`** takes a matrix and a vector and return LU matrix, piv vector implying the permutation, the result x and an integer indicating the status.

In fact, in the implementation, **`DGESV`** might call `DGETRF` and `DGETRS`, which do LU decomposition and find the solution respectively.

3.1 Try different matrix sizes and measure/plot their running time

I tried matrices of size $N = i \times 10^3$, $\forall i \in [1, 20] \cap \mathbb{N}$. For each size the matrices were randomly generated and tested 10 times so as to take the average.

The running time of each step is measured by `pref_counter` came with Python time module, which may provide higher resolution. Even though the time to solve the largest matrix is about 5 seconds, the total time the job totally consumed was quite longer. I guess it may be due to the random processes took the most of time.

The Python script was ran on a single full node of **Taiwania 3** which is equipped with 2 Intel® Xeon® Platinum 8280 CPU sockets and 56 CPU cores in total.

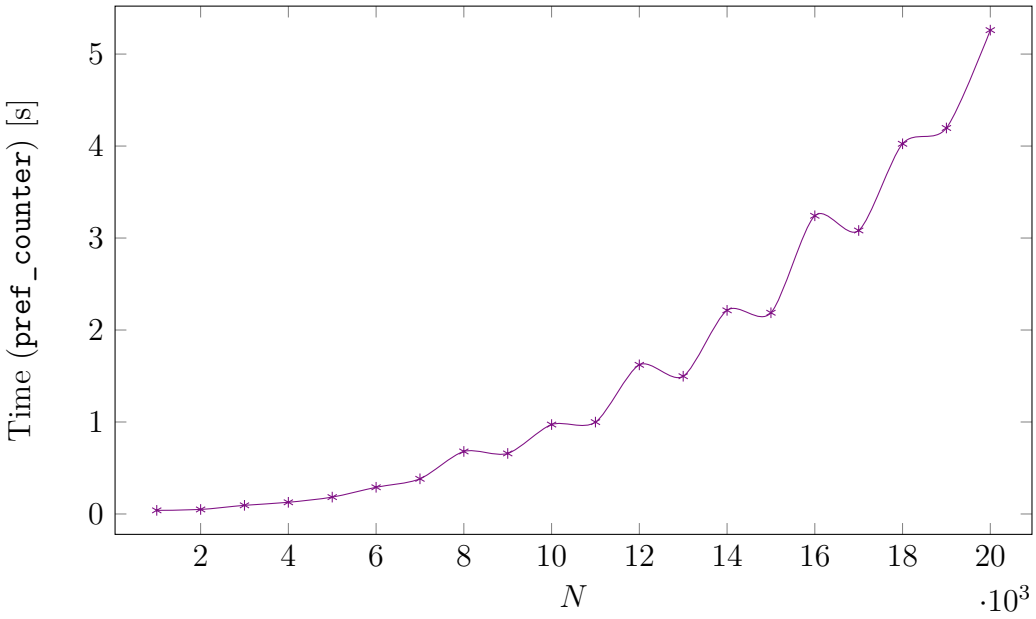

Figure 1: Time in seconds measured by `pref_counter` of different matrix size N

The outcome is illustrated in Figure 1. The curve looks like a parabola, but it has a tendency to drop a bit or reach somewhat local minima at $N = i \times 10^3$ when $i > 8$ and i is odd. This result is quite "odd", which really astonished me.

3.2 How to verify the correctness of the solution EFFICIENTLY?

We could just substitute x and check whether the dot product Ax is equivalent to b. Note that since there are always inevitable **floating-point errors**, we couldn't use the "==" operator directly. Instead, we should check if all differences of corresponding elements is less than a tolerance threshold, or so-called ϵ. In `numpy`, there is `np.allclose()` function to help us realize this elementwisely. In fact, my assertion `np.allclose(np.dot(A, x), b)` failed once where $N = 2 \times 10^4$ then, which might because the matrix was too large that the **floating-point error** accumulated and exceeded ϵ.

4 Solve linear systems via block matrices

> "The reason why BLAS/LAPACK can be so effective on modern CPUs is that they utilize block matrix operation."

4.1 How to use block matrix operation to solve a linear system $Ax = b$?

> "Derive the formula by partitioning A into a 2×2 block matrix."

Based on the above hint provided by TA, we could partition A, x, b so that $\begin{pmatrix} A_{0,0} & A_{0,1} \\ A_{1,0} & A_{1,1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \end{pmatrix}$. If we would like to solve this, we could exert LU decomposition for block matrix.

In the section 3, I discovered that it looks like that LAPACK do the PLU decomposition. Nevertheless, I only found LDU decomposition for block matrix:

$$\begin{pmatrix} A_{0,0} & A_{0,1} \\ A_{1,0} & A_{1,1} \end{pmatrix} = \begin{pmatrix} I & O \\ A_{1,0}A_{0,0}^{-1} & I \end{pmatrix} \begin{pmatrix} A_{0,0} & O \\ O & A_{1,1} - A_{1,0}A_{0,0}^{-1}A_{0,1} \end{pmatrix} \begin{pmatrix} I & A_{0,0}^{-1}A_{0,1} \\ O & I \end{pmatrix}$$

. As a consequence, we could find the result of each entry of x_0, x_1 recursively.

4.2 Explain why a good matrix-matrix multiplication subroutine can accelerate the computation of solving $Ax = b$

Just as shown above, we could see that we would need a lot matrix multiplications while factoring the LDU. In particular, the **time complexity** could be expressed recursively as $T(n) = T(\frac{n}{2}) + 4F(n) + O(n^2)$, where $F(n)$ is the **time complexity** of matrix multiplication and $O(n^2)$ is the one of matrix addition.

So let's discuss on the result of $T(n)$ of some $F(n)$ by **Master Theorem**[1]:

$F(n) = O(n^3)$ This is the case of naïve approach. Then $T(n) = T(\frac{n}{2}) + O(n^3) = O(n^3)$.

$F(n) = O(n^{\log_2 7})$ This is the case of Strassen's algorithm in use. Then $T(n) = T(\frac{n}{2}) + O(n^{\log_2 7}) = O(n^{\log_2 7})$.

By observation, I found that since $\log_2 1 = 0$, $T(n)$ would be equal to $F(n)$ in the most cases so long as the degree of $F(n)$ is greater than 0 and 2. Therefore, the better we could do matrix multiplication, the better we could solve the linear system.

5 What is the Strassen Matrix-Matrix Multiplication algorithm?

We are to calculate $C = AB$, where A, B are square matrices. Without lost of generality, if the size of matrices is not a power of 2, we could add rows and columns with 0s.

Then we could adopt the *divide-and-conquer* strategy and partition the matrices into block matrices: $A = \begin{pmatrix} A_{0,0} & A_{0,1} \\ A_{1,0} & A_{1,1} \end{pmatrix}, B = \begin{pmatrix} B_{0,0} & B_{0,1} \\ B_{1,0} & B_{1,1} \end{pmatrix}, C = \begin{pmatrix} C_{0,0} & C_{0,1} \\ C_{1,0} & C_{1,1} \end{pmatrix}$, where $A_{i,j}, B_{i,j}, C_{i,j}$ are matrices of size 2^{n-1}.

[1] Just in case if needed:

$$T(n) = aT(\frac{n}{b}) + O(n^d) = \begin{cases} O(n^d), & d > \log_b a \\ O(n^d \log n), & d = \log_b a \\ O(n^{\log_b a}), & d < \log_b a \end{cases}$$

The naïve algorithm would be $\begin{pmatrix} C_{0,0} & C_{0,1} \\ C_{1,0} & C_{1,1} \end{pmatrix} = \begin{pmatrix} A_{0,0}B_{0,0} + A_{0,1}B_{1,0} & A_{0,0}B_{0,1} + A_{0,1}B_{1,1} \\ A_{1,0}B_{0,0} + A_{1,1}B_{1,0} & A_{1,0}B_{0,1} + A_{1,1}B_{1,1} \end{pmatrix}$.
Strassen found that the following 7 matrices:

$$\begin{aligned} M_0 =& (A_{0,0} + A_{1,1})(B_{0,0} + B_{1,1}) \\ M_1 =& (A_{1,0} + A_{1,1})B_{0,0} \\ M_2 =& A_{0,0}(B_{0,1} - B_{1,1}) \\ M_3 =& A_{1,1}(B_{1,0} - B_{0,0}) \\ M_4 =& (A_{0,0} + A_{0,1})B_{1,1} \\ M_5 =& (A_{1,0} - A_{0,0})(B_{0,0} + B_{0,1}) \\ M_6 =& (A_{0,1} - A_{1,1})(B_{1,0} + B_{1,1}) \end{aligned}$$

such that $\begin{pmatrix} C_{0,0} & C_{0,1} \\ C_{1,0} & C_{1,1} \end{pmatrix} = \begin{pmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 - M_2 + M_3 + M_6 \end{pmatrix}$.

5.1 Why it is faster than the conventional matrix-matrix multiplication algorithm in theory?

If we follow the definition to do the matrices multiplication, it's obvious that the **asymptotic time complexity** would be $O(n^3)$.

After partitioning the matrices into block matrices and applying the *divide-and-conquer* approach, we would have the recursion expression of **time complexity**

$$T(n) = 8T(\frac{n}{2}) + O(n^2)$$

since we would have to do 8 smaller block matrices multiplications and the time to do matrices addtiions is $O(n^2)$. By the **Master Theorem**, we could solve that

$$T(n) = O(n^{\log_2 8}) = O(n^3)$$

, which is the same as original.

Yet in Strassen's method, we only need to do 7 multiplications. Hence the recursive **time complexity** becomes

$$T(n) = 7T(\frac{n}{2}) + O(n^2)$$

. So by the **Master Theorem** again, we have

$$T(n) = O(n^{\log_2 7}) \approx O(n^{2.8})$$

, which is the first matrices multiplication algorithm ever that beats the definition.

5.2 Why people usually do not use it in practice for high performance computation?

Although nowadays there are some fancy algorithms that could do matrices multiplication in even $O(n^{2.3})$ approximately, most of them are not so practical at all since they're often too complex and with large constants in their **time complexity**. Still, Strassen's algorithm is of use in BLAS/LAPACK to some extent.

When it comes to high-performance computing, since that the original definition approach is in a form that is easy to be parallelized by MPI (Message Passing Interface) or OpenMP (Open Multi-Processing) across nodes or threads respectively, sometimes it might even outperform Strassen's algorithm or other sub-$O(n^3)$ algorithms.

Acknowledgements

I thank to National Center for High-performance Computing *(NCHC)* for providing computational and storage resources.